A Collaborative Go Playground using Replicated Growable Array CRDTs

CPSC 416 Project 2 Report

Eduard Kagrimanyan m9r8@ugrad.cs.ubc.ca
Nathan Su b0z8@ugrad.cs.ubc.ca
Tim van der Kooi b4n0b@ugrad.cs.ubc.ca
Michael Yau i5n8@ugrad.cs.ubc.ca

ABSTRACT

In distributed collaborative editors, high responsiveness is essential in maintaining a seamless and enjoyable user experience. This is usually achieved using remote data replication - however, such methods introduce the challenge of maintaining consistency across replicas. Using *conflict-free replicated data types* (CRDTs), one can achieve strong *eventual* consistency, which may be sufficient for most collaborative editing use cases. The *GoLab Playground* is a collaborative code editor for the programming language Go, relying on a class of sequence CRDTs known as *Replicated Growable Arrays*. The system allows for concurrent editing and execution of Go programs in remote sandboxed environments, as well as the ability to save and reload sessions which persist on a simple distributed file system.

OVERVIEW

1 INTRODUCTION

In the past decade, distributed collaborative editors have given users the ability to share, access, and edit files simultaneously with other users all across the world. Similarly, collaborative code editors allow groups of developers to share and edit code in real-time. With features such as syntax highlighting, autocompletion, and the ability to quickly start up and share new editing sessions, they provide an excellent resource for interviews, debugging, teaching, and more.

Due to the unavoidable nature of network latency, good performance in a collaborative editing system usually requires that each user maintains their own copy, or replica, of the shared document. Any user's local changes are propagated to other users, during which the system uses one of many techniques to maintain consistency across all replicas, such that all users will apparently be editing the same document at any given time. The widely used Google Docs employs *operational transformation* (OT), an approach that relies on a centralized server to handle conflicting operations. On the other hand, conflict-free replicated data types (CRDTs) are a decentralized solution to concurrency control, and alleviate many bottlenecks and single points of failure associated with OT. CRDTs (and a subset known as *operation-based* CRDTs) guarantee strong eventual consistency, meaning that replicas will converge to the same state given enough idle time in the system.

The *GoLab Playground* is a collaborative code editor for the programming language Go and utilizes a class of sequence CRDTs known as Replicated Growable Arrays (RGAs). It is a responsive, scalable solution for concurrent editing and execution of Go programs, which can be compiled, linked, and run inside of remote sandboxed environments. The system is supported by a network of worker nodes which are robust to failure and seamlessly allows for concurrent sessions which can be saved and reloaded at any time. The GoLab Playground is also a proof-of-concept that demonstrates the potential of RGA CRDTs in the space of collaborative code editors, and can be easily extended to support more programming languages in the future.

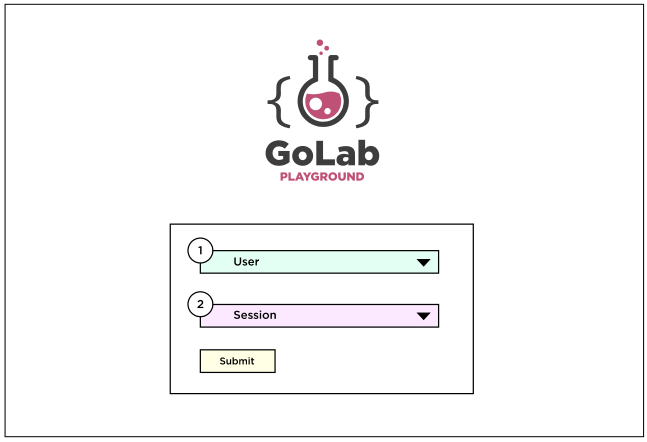

Figure 1: Mockup of the login interface

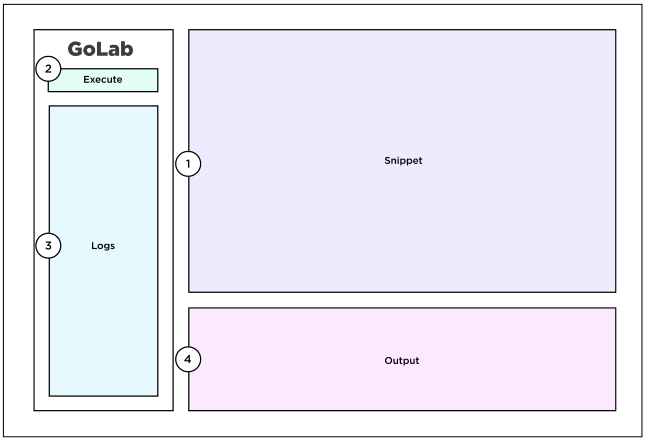

Figure 2: Mockup of the editor interface

2 DESIGN

2.1 Playground Interface

The GoLab Playground consists of two main views: a *login* and an *editor* interface. Upon entering the login page (Figure 1), the user will be prompted with the option to choose an existing identity or create a new one (1). There are no additional login credentials. The user can then choose to enter an existing session or create a new one (2). Clicking submit will open the session.

When the editor (Figure 2) is opened, the source Go code for that session is displayed in the *Snippet* panel (1). The user can run (a single step for compiling, linking, and running) the program by clicking the execute button (2). As soon as the code is run, a new log will appear in the *Logs* panel (3), which can be clicked to reveal the terminal output in the *Output* panel (4) generated by that specific execution. The original source code which was run will also be displayed in the Snippet panel (as read-only). The user can exit back to the editor at any time.

2.2 System Components

2.2.1 Application Layer

Client

A client is a browser session connected as a user in the system. The browser session provides an interface for the user to edit, compile, and run code snippets with other users. Each newly created snippet generates a session ID which can be shared with collaborators who wish to edit that same snippet.

When a user connects to the application server (described below), they are presented with the application which receives the IP of the dedicated worker node (see section on Worker Layer below). The browser then establishes a WebSocket connection to the worker.

Application Server

The application server is responsible for bootstrapping a new client by delivering necessary HTML and Javascript. When a client connects, it retrieves the IP of a worker node from the load balancer (see section on Worker Layer below) for a client. The application server is also serves to verify session ID uniqueness whenever users attempt to create new sessions.

This server must be run in Go such that it can make RPC calls to the load balancer, otherwise the load balancer would need to have REST endpoints.

2.2.2 Worker Layer

Worker Node

A worker node is responsible for disseminating *operations* (see *2.3.2 - Editing*), maintaining a local copy of any number of code snippets (sessions), and compiling and executing code snippets. The worker performs a simple flooding protocol for editor operations: when a worker receives an operation from a client or worker, the worker disseminates the operation to its connected workers. These operations are saved locally and periodically sent to all connected clients on a session-specific basis. Active sessions are periodically saved to the distributed file system.

Workers can also be selected by the load balancer to complete a job to compile and run any code snippet in the system. The node will maintain a queue of jobs should it receive more than one job to execute. This is described in further detail in *2.3.4 – Code Execution.*

Load Balancer

The load balancer is responsible for managing all the worker nodes in the worker layer. It balances client-to-worker load between all workers, manages worker-to-worker connectivity by attempting to maintain some pre-defined network topology, and maintains a record of active workers and their connected clients using heartbeats.

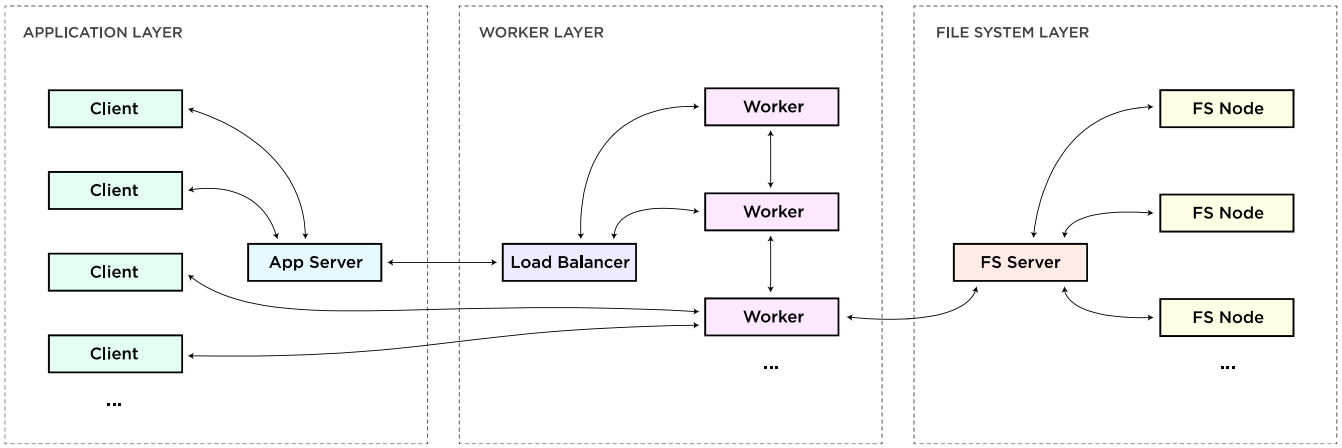

Figure 3: System components

The load balancer receives requests from the app server whenever a new user wishes to join the system and needs to be assigned a worker node. The load balancer also manages code execution requests - when it receives such a request, it will choose a worker to compile and run the snippet, after which worker will return the execution log. This log is then disseminated to all connected workers.

When the load balancer receives a request from a user to join the system or execute some snippet of code, the load balancer will always choose a worker which has the fewest connected clients.

2.2.3 File System Layer

File System Server

The file system server is responsible for managing a simple distributed file system comprised of a network of file system nodes. It maintains a directory of editable code snippets, encoded as sequence CRDTs and identified by session ID, as well as all job requests, corresponding to saved code snippets ready to be compiled and their associated execution logs, identified by job ID.

When a write request is received, an attempt is made to save the payload at all connected file system nodes. Each save attempt that is positively acknowledged will result in the directory being updated such that the node becomes a known replica for the saved data. When a read request is received, nodes which are known replicas are picked at random to attempt to retrieve the data - the request will only fail after all known replicas fail to retrieve the data.

File System Node

The file system node is responsible for handling read and write operations of sessions and logs. They act as replicas for all persistent data in the system. Their identity, which is assigned by the file system server, is stored on disk such that failure recovery is possible (see further sections on failure handling).

2.3 System Operation

2.3.1 Setup

When a new user joins the system, they must connect to a worker node in order to participate in collaborative editing - this workflow is described in Figure 4. A request to create or join an existing session (1) will be sent to the app server, which will ask the load balancer (2) in the worker layer for the IP of an available worker node. This IP will be forwarded back to the client (3) (4), after which the client can establish a WebSocket connection with that worker (5). All subsequent communications from this client to the system will be through its connected worker.

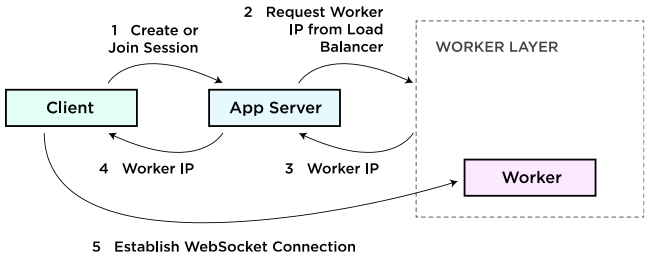

Figure 4: Client-worker setup

2.3.2 Editing

Client

In order to avoid reinventing the wheel (and to provide a visually appealing editor interface), a third-party JavaScript library called *CodeMirror* was used to not only format the code but also provide a good starting point for "hooking" into local inputs and applying remote inputs. With that said, the editing behaviour driven by two types of editing operations: *local operations* and *remote operations*.

Local operations are operations performed by the user locally. Whenever the user performs an operation, it is

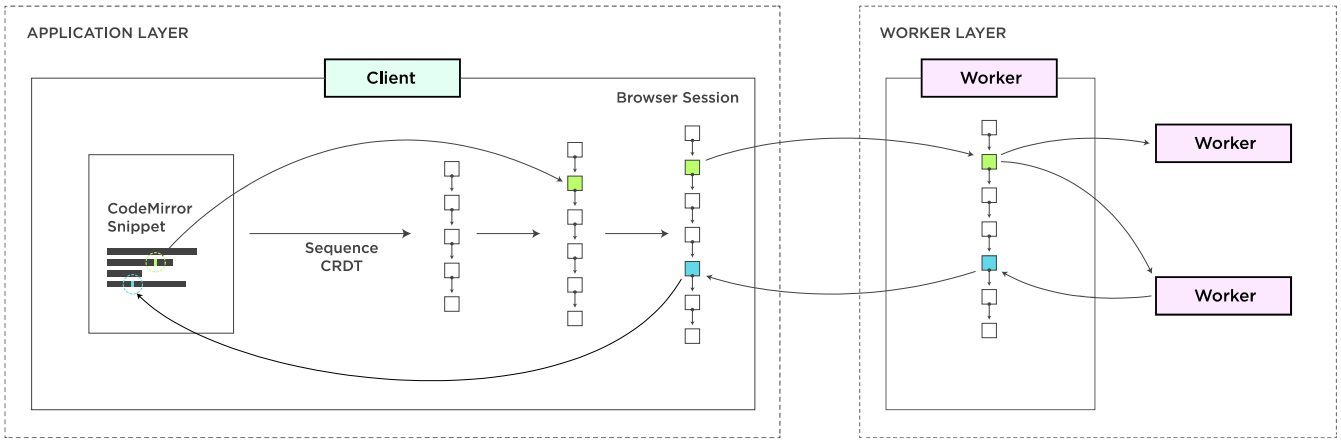

Figure 5: Client-worker operation flow

applied to its local sequence CRDT and sent over WebSocket to its assigned worker.

Remote operations are operations performed by other users, which the worker receives and sends over WebSocket to the client. When a remote operation is received it is applied to the local sequence CRDT and the editor text is updated accordingly.

Given that the CRDT is only concerned with IDs, a data structure was needed that could map IDs to positions in the actual document. As such, a 2D array was used, where each row of the array maps to a line in the document and each element in the row maps to a character in the line. Using this mapping, anytime the local user performs an input or a delete, CodeMirror's API is used to get the position of the operation and subsequently reference the mapping to either determine the previous ID or get the element ID. Similarly, when a remote operation is received, the previous ID can be used to reference the CRDT and the mapping to determine what position to apply the operation.

Worker

Once connected to the worker, the client can begin to edit their specified session. Clients can add/delete characters, copy/paste, and undo/redo previous operations. As soon as the client begins to edit, modifications are made to the client's CRDT to reflect their local changes, which are then passed on to the worker's CRDT to update other connected clients' sessions and resolve potential insertion conflicts in the CRDT if other users are editing in the same place concurrently (Figure 5).

Since several clients can be connected to the same worker, a worker will first insert all of their clients' local operations into the CRDT that corresponds to the edited session. Each CRDT is a hashed doubly-linked-list, so each incoming operation only needs to know the element that comes before it. The insertion algorithm of an incoming operation will look up the ID of the previous element, and then change that element's next ID to the incoming operation's ID, as shown in Figure 6.

Other workers need these operations to update their own replica of the session's CRDT, so each worker will store two seconds worth of connected client operations before sending them to all of their connected workers. Once the sending worker receives an acknowledgement from another receiving worker, the sending worker will clear their local operations cache for the next batch. When a worker receives these operations, the worker pretends that these operations are local and inserts them in the same manner as the insertion algorithm mentioned above. If some of these operations already exist in their CRDT, they will drop them from their local operations cache and not send them to other workers. This flooding protocol ensures that workers will see all operations that pertain to the sessions they handle.

With all of these clients connected and editing concurrently, it is necessary to ensure that each replica CRDT will eventually be in the same state to have eventual consistency. A CRDT can fall out of sync when two or more clients try to insert an element in the same position. For example, multiple clients may try to place an element right after the same previous ID at the same time, and this could result in inconsistencies across these clients because each worker will put their local clients' operations in first.

To resolve this issue, the worker node uses an algorithm that compares element IDs to reach eventual consistency. An incoming element will ideally place itself after it's targeted previous ID, but it will first look at the previous ID's next ID in that CRDT. If the next ID is greater than the incoming element's ID, then the incoming element will try to place itself after the next ID's element. It will continue to do this until it reaches a next ID that is not greater than itself. This algorithm guarantees eventual consistency as all incoming operations go through this same process.

2.3.3 *Sessions*

Since clients can start new sessions or join sessions that have been started by other clients, workers must also

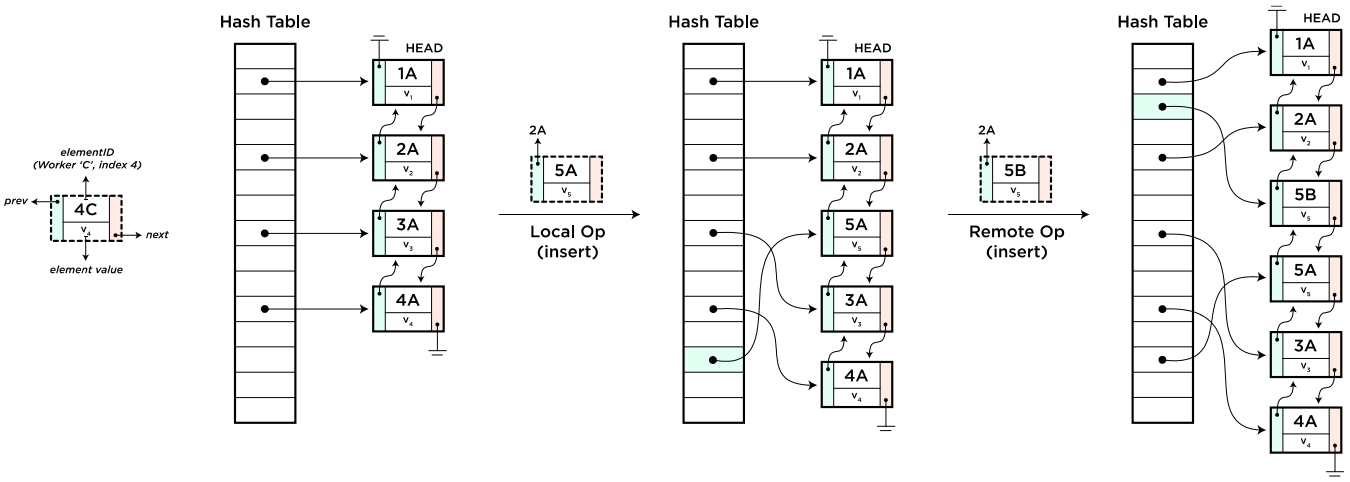

Figure 6: Example RGA structure

handle sessions for the client. The worker will pass the session to the client, which contains log outputs of previously executed code in that session and the session's current CRDT.

When a client asks for a given session, a worker will first check to see if it already has the session. If it doesn't, that worker will ask its connected neighbour workers to retrieve the session. If none of these workers have the session, then the worker will have to contact the file server to retrieve the session.

2.3.4 Code Execution

When users are ready to run their code, the application goes through a protocol that involves each layer of our system architecture. This protocol is detailed in a sequence diagram (Figure 7) and the follow sections.

Initialization

When the user will presses the execute button in the application, its connected worker nodeis notified. The worker node will generate a job ID and save the current code snippet to the file system as a pending job. It then passes the job ID to the load balancer, which will choose a worker node to run the job.

Execution

The load balancer will pass the job ID to a worker node to compile and run the code. The selected worker node will use the job ID to retrieve the code snippet from the file system - or if the code has already been compiled and executed previously, it will receive the execution log for that compiled code. If the code has not been executed, the worker node will save and run the code on its machine locally. If the code does not compile, any error(s) returned will be reflected in the log.

Results

Once the code is run, the worker node will save the log output on the file server with the job ID. It will also send the log to the load balancer to broadcast to all connected worker nodes, which will then send the log to all of their connected clients.

2.4 Failure Handling

2.4.1 Worker Failures - Editing

When a worker fails, both workers and clients can be affected by operation loss because connected workers will not receive the failed workers' operations and the clients connected to the failed worker will not receive operations. These failures are handled by utilizing caching on both the worker and client, as well as transparent recovery of the client on worker failures.

The client cache is an ordered array of local operations that have not yet been ACKed by the worker. After the client handles a local operation, it will push the operation to the cache and only remove it from the cache when the worker responds with that very same operation. On the other hand, when the worker receives operations from its clients it will maintain its own array of operations that must be ACKed back to the client, such that it will not ACK until the operation is sent to the minimum required number of connected workers.

The worker cache is an ordered array of operations, which is a record of all operations received in the last 10 seconds from clients and workers. The purpose of this cache is strictly for client reconnection, such that when a worker fails and the client connects to a new worker, the client will request the cached operations. Note: it is assumed that 10 seconds is roughly twice the amount of time necessary for the client to fully recover, so it is unlikely that 10 seconds is insufficient to sync the client.

Now, with this framework in place, when the worker fails the client reconnects to a new worker, where both the client and the new worker will exchange their cached operations to re-sync with each other. As long as the client

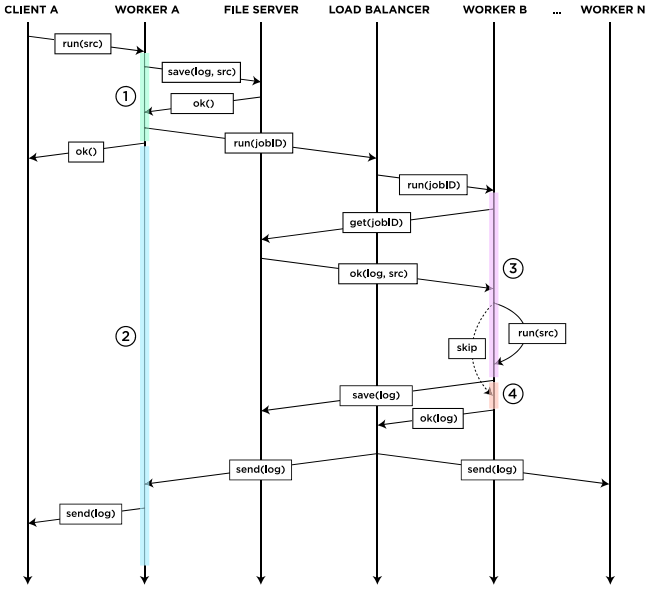

Figure 7: Sequence diagram for the code execution protocol and potential regions of failure

does in fact reconnect, it is highly unlikely for the client to fall out of sync or for the clients operations to get lost.

2.4.2 Worker Failures - Code Execution

Throughout the protocol described in Figure 7, there are several points of failure that are handled appropriately by our system. There are four major regions of failure that are marked with the numbers (1) to (4). These error cases are handled as described in the following sections.

1. As previously mentioned in *2.3.3 - Sessions*, the Load Balancer is in charge of code execution after a worker has generated and saved the job when a client issues an execute command. If Worker A fails at any time before sending an ok() back to Client A, that client will reconnect to a new worker node via the load balancer and resend the run() command to that new node. Client A will know if Worker A fails because they will utilize a heartbeat mechanism to assert connection status.
2. If Worker A fails after sending an ok() back to Client A, that client will reconnect to a new worker node but will not resend the run() command to the new node. It will not resend the command since the load balancer will find some worker node to complete the job, and will also distribute the results of that job to every worker node. The client will eventually receive the execution log.
3. If Worker B fails any time between receiving the run() request and sending the final log output to the file server, the load balancer will find a new worker node and call run() on that node.
4. If Worker B compiles and runs the code, saves the log output to the file server, but fails to send log output back to the load balancer, the load balancer will assume the worker has failed. It will then ask a new worker node to run the code snippet, but that worker node will find out that the snippet has already been executed when it receives a log from the file server instead of the snippet source. Instead of running the job again, it can proceed to return the log to the load balancer.

If all worker nodes fail, all clients will not be able to find any worker node to reconnect. In this case, the system will no longer be operational.

2.4.3 File System Node Failures

The file system nodes are fault tolerant and can fail and rejoin the network as long as their nodeID is on disk. The file system server keeps track of the latest version on each of the nodes and will only retrieve the most up to date version of sessions or logs. However, if all file system nodes fail, sessions and logs cannot be saved or retrieved.

2.4.3 Other

Client browsers may disconnect and reconnect at anytime, but will need to rejoin the session again to continue editing. This does not impact the rest of the system in any way.

3 IMPLEMENTATION

3.1 System Backend

All of the system back-end has been written in Go and all the nodes in the system communicate over the RPC protocol. Node-to-node communications are restricted to the following:

Application Server	$\rightarrow$	Load Balancer
Load Balancer	$\leftrightarrow$	Worker
File System Server	$\rightarrow$	File System Node
Worker	$\leftrightarrow$	Worker
Worker	$\rightarrow$	File System Server

Furthermore, both the application server and worker nodes have HTTP endpoints. The application server provides the client with the web content through its endpoint and the Worker provides two services through its single endpoint: an entry point for an incoming WebSocket connection from the client and an endpoint for the client to open, close or recover its session.

3.2 Deployment

The application was deployed on 10-15 VMs hosted on Microsoft Azure. Ansible, a configuration management and application deployment tool was used to automate deployment and teardown of the application on Azure. With an entrypoint VM hosting the Ansible program to contact all of our allocated VMs, it became easy to create more worker and file system nodes to be utilized in the application, as well as simulate failures with a few simple Ansible commands from the entrypoint machine.

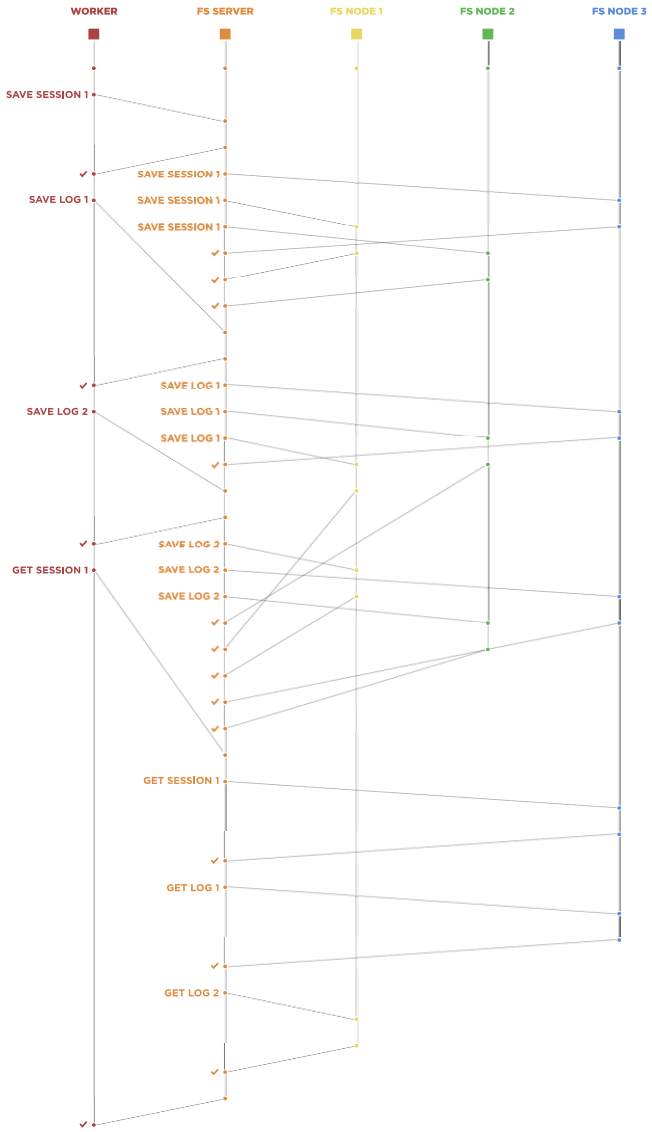

Figure 8: ShiViz graph of file system operation

4 IMPLEMENTATION

4.1 Testing

Testing included both manual testing and studying GoVector logs, but we didn't get a chance to implement unit and integration tests due to time constraints. Furthermore, we also decided against end-to-end testing as it would require complicated JavaScript test suites that would have to be coordinated with other clients. In general, we found that manual testing sufficed for our purposes given that we have a clear test plan to execute.

Our testing plan includes failure-free cases, worker node failures, CRDT consistency between clients and workers, and code execution.

Worker node failures were tested with various cases also mentioned previously. Worker nodes were failed before and after executing a snippet to ensure a code snippet was eventually executed. When executing a code snippet, a worker node will also set a timeout in order to prevent infinite loops from consuming excess CPU resources. Worker nodes were failed when connected to a client to test the recovery of a client in reconnecting to another worker node. When a worker node reconnected, testing was done to observe that clients utilize the reconnected worker.

CRDT consistency was tested between both clients and workers. This involved testing various insert and delete operations in same or different places and then to observe the eventual consistency of all the CRDTs.

Code execution testing followed the failure cases of *2.4.2 – Worker Failures – Code Execution* to ensure correct recovery and behaviour

4.2 ShiViz

ShiViz-compatible vector-clock timestamped logs were created by instrumenting system RPC calls using *GoVector*. These were used to visualize two example workflows proposed in the original design. The original logs used to generate these visualizations can be found in the project repository.

Figure 8 shows a communication graph generated using ShiViz (with additional annotations for improved clarity). This workflow consists of a single worker, a file system server, and three file system nodes. The worker begins by saving a session to the file system by sending it to the file system server. The server then distributes the session to its connected file system nodes, who write the session to disk. Afterwards, the worker saves two logs associated with the previously saved session, and then attempts to retrieve the session (which will grab the associated logs) from the file system. When this request is made, the file system server will pick nodes at random to retrieve the session and logs - and this is in fact what happens, as the two logs are retrieved from different nodes even though we see that all three nodes have both logs saved.

Figure 9 (another ShiViz graph) demonstrates a code execution workflow in a system containing two worker nodes. First, a log (containing the code snippet) is saved to the file system. A request is then made to the load balancer to run the job, who then finds a worker node to do so. We see that *worker 2* receives this request, who then retrieves the source from the file system, executes the code, and saves the log. Overall, these graphs support a correct implementation of two workflows according the proposed design of the system.

5 DISCUSSION

5.1 Challenges

CRDT Implementation

It was difficult to implement a consistent CRDT in our web application due to different requirements from the backend and the frontend. Whereas the backend was necessary to distribute the CRDT to all workers and implement eventual consistency to resolve conflicts from concurrent input across clients, the frontend also required it's own CRDT implementation to handle local and remote input quickly to reduce latency and ensure that the content in the editor window was in sync with a session's CRDT.

Our problem was a matter of efficiency: by having a CRDT implementation in the backend and the frontend, our application appeared to be performing twice the amount of work. It wasn't possible to place the CRDT logic exclusively in the frontend because it wouldn't be able to resolve conflicts from concurrent input. Putting the CRDT exclusively in the backend would have caused extreme latency on the frontend, which would have to reload the CRDT content into the editor window upon every update.

5.2 Complexity and Scalability

We went with a centralized load balancer and file system server which gave our system two single points of failure. This design also somewhat limits scalability as all worker nodes and file system nodes need to have a connection to their respective servers.

The file system kept a simple distributed implementation to avoid overcomplicating the project since most of the importance was on the CRDT implementation.

The worker topology used was the same as the given from Project 1 in the config.json. This reuse was to simplify the load balancer implementation as the worker nodes used a flooding protocol in disseminating operations.

5.3 Limitations

Manually testing on different machines and in different environments was a limitation to our team's ability to iterate and make changes to our code. Since our production environment is on Azure, some parts of our code had to be specifically tailored to work in that environment, such as resolving public IP addresses, RPC timeouts, and resolving hostnames for more detailed logging. Testing these changes on Azure required IT automation to speed up the cycle time of testing, but the time invested in providing automation limited the time spent on developing the application.

5.4 Distribution of Work

Eduard worked on the implementation of the client and client-worker communication. More specifically: design of the UI, implementation of the editor/CRDT logic in the client, worker and client caching, and client recovery.

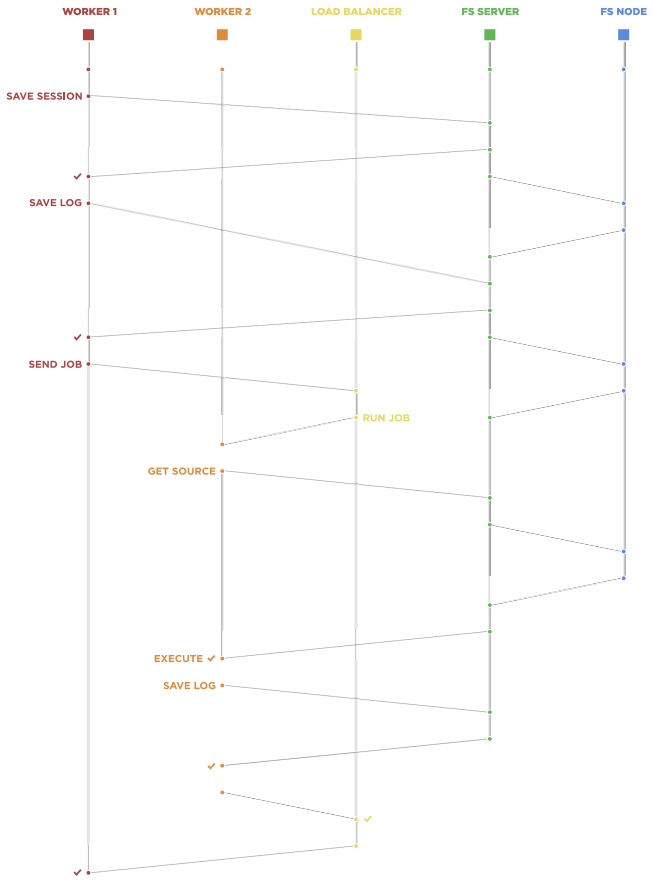

Figure 9: ShiViz graph of code execution

Tim worked on much of the worker implementation. More specifically the CRDT logic on the back-end for applying operations to the CRDT and disseminating operations, and worker-to-worker setup. Furthermore, Tim did all the Azure work for spinning the system up, taking it down, and logging among all the VMs.

Nathan did a lot of the foundational work such as the Application server, the Load Balancer and the client-to-worker WebSocket implementation. Furthermore, he was responsible for implementing the logic for code execution, as well as the accompanying logic for the logs at the clients.

Michael implemented the file system and incorporated GoVector into RPC communications to generate ShiViz-compatible logs. He also created graphics, all the figures, and formatted the report. Michael also drew this gopher:

6 REFERENCES

[1] https://hal.inria.fr/inria-00336191/PDF/main.pdf
[2] https://hal.archives-ouvertes.fr/file/index/docid/921633/filename/fp025-nedelec.pdf
[3] https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type
[4] https://www.youtube.com/watch?v=9xFfOhasiOE
[5] https://www.microsoft.com/en-us/research/video/strong-eventual-consistency-and-conflict-free-replicated-data-types/
[6] https://martin.kleppmann.com/2016/11/03/code-mesh.html
[7] https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html
[8] http://simongui.github.io/distributed-systems/crdt.html
[9] https://github.com/netopyr/wurmloch-crdt
[10] https://bestchai.bitbucket.io/shiviz/
[11] https://github.com/DistributedClocks/GoVector